二叉树的存储、遍历和其它操作

二叉树的抽象数据类型定义

ADT BiTree

DataModel

二叉树由一个根结点和两棵互不相交的左右子树构成，结点具有层次关系

Operation

- BinaryTreeNode(): 创建一个二叉树结点
- CreatBinaryTree(*value*, *left_tree*, *right_tree*): 构造二叉树，根结点的数据为*value*，左子树和右子树分别是*left_tree*和*right_tree*
- IsLeaf(*tree*, *node*): 如果二叉树*tree*中结点*node*为叶结点，返回**true**；否则返回**false**
- Height(*tree*): 返回二叉树*tree*的高度（深度）
- PreOrder(*tree*): 前序遍历二叉树*tree*
- InOrder(*tree*): 中序遍历二叉树*tree*
- PostOrder(*tree*): 后序遍历二叉树*tree*
- LevelOrder(*tree*): 层序遍历二叉树*tree*

二叉树的操作

endADT

学习目标

- 掌握二叉树的存储结构（顺序和二叉链表）
- 了解三叉链表
- 熟练掌握二叉树的遍历操作（前序、中序、后序、层次）
- 掌握二叉树的递归遍历算法，熟悉非递归算法
- 二叉树的创建、销毁和其它操作

顺序存储

顺序存储结构的要求是什么？

用一组连续的存储单元依次存储数据元素，由存储位置表示元素之间的逻辑关系

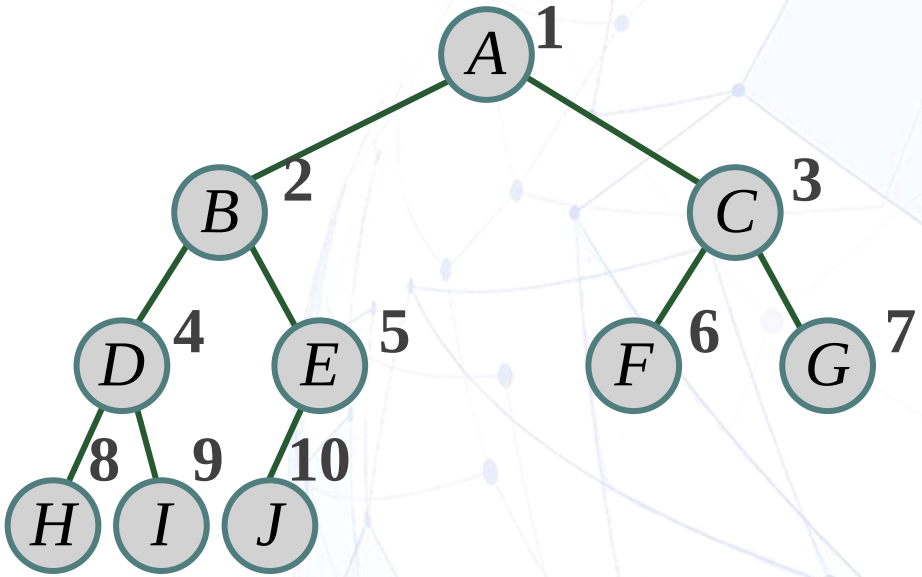
二叉树的顺序存储结构是用一维数组存储二叉树的结点，结点的存储位置（下标）应能体现结点之间的逻辑关系——父子关系

如何利用数组下标来反映结点之间的逻辑关系？

完全二叉树中结点的编号可以唯一地反映结点之间的逻辑关系

顺序存储

如何定义二叉树的顺序存储结构呢?



以编号为下标

0	1	2	3	4	5	6	7	8	9	10
	A	B	C	D	E	F	G	H	I	J

```
const int MaxSize = 100;
template <typename DataType>
class SeqBiTree
{
public:
    SeqBiTree( );
    ~SeqBiTree( );
    void PreOrder( );
    void InOrder( );
    void PostOrder( );
private:
    DataType data[MaxSize];
    int biTreeNum;
};
```

顺序存储

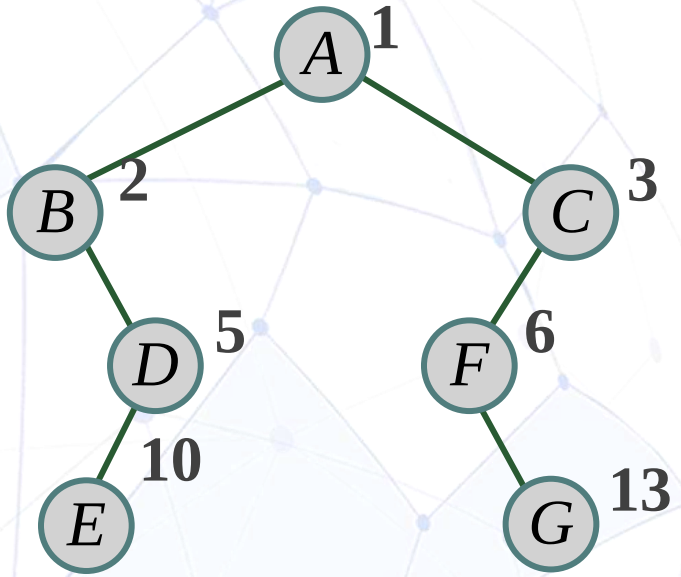
对于普通的二叉树，如何顺序存储呢？

将二叉树按完全二叉树编号：

（1）根结点的编号为 1

（2）若某结点 i 有左孩子，则其左孩子的编号为 $2i$

（3）若某结点 i 有右孩子，则其右孩子的编号为 $2i+1$



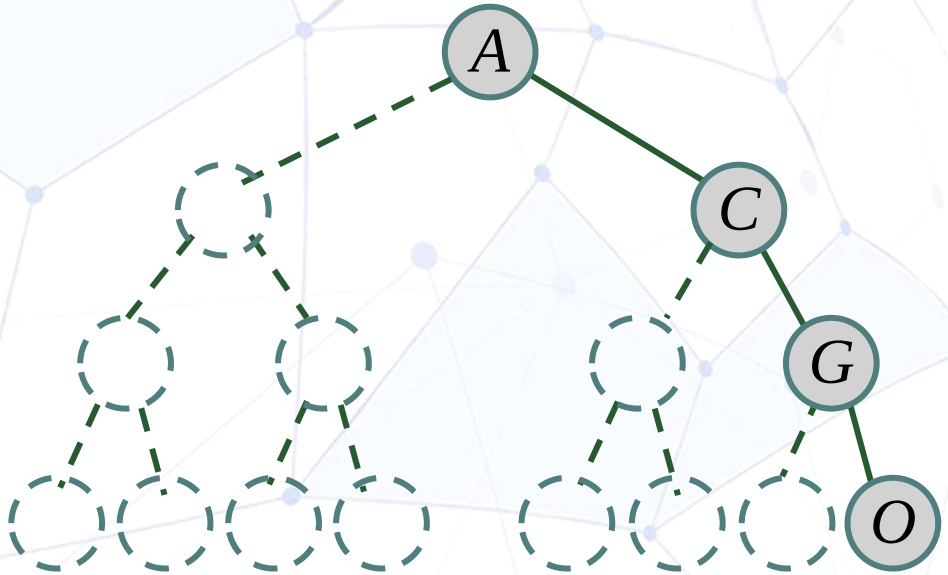
以编号为下标

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	∧	D	F	∧	∧	∧	E	∧	∧	G

顺序存储

顺序存储一棵右斜树会发生什么情况?

缺点：浪费存储空间



二叉树的顺序存储结构一般仅存储完全二叉树

二叉树的顺序存储实现

优点

(1) 只需用顺序表存放结点数据，不需要保存结点间逻辑关系

(2) 结点间逻辑关系可通过相对位置确定

(3) 对于顺序存储结构第 k 个位置的结点 ($k \geq 1$)，其左右子结点分别存储在第$2k$和第$2k+1$个位置，父结点在位置$\lfloor k/2 \rfloor$。因此查找子结点和父结点只需O(1)的时间

(4) 顺序结构是完全二叉树最简单、最节省空间的存储方式，n个结点只需O(n)的空间

缺点

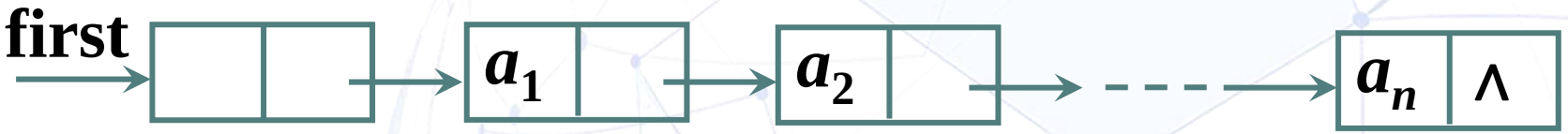
(1) 对一般的二叉树，可能造成空间浪费

(2) 在最坏情况下(右斜树)，存放n个结点的二叉树可能需要长度为$O(2^n)$（右斜树深度为n，化成完全二叉有2^n-1个结点 ） 的顺序表

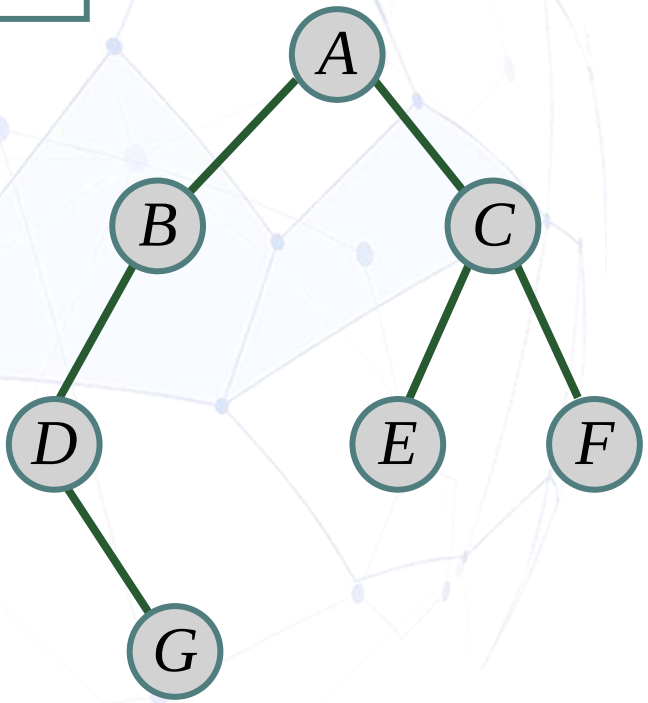
(3) 可用其它序列化的方法降低顺序存储的空间复杂度，但无法通过在顺序表中的相对位置直接确定两个结点是否有父子关系，增加了查询结点间逻辑关系的时间复杂度

二叉链表

如何用链接存储方式存储二叉树呢?

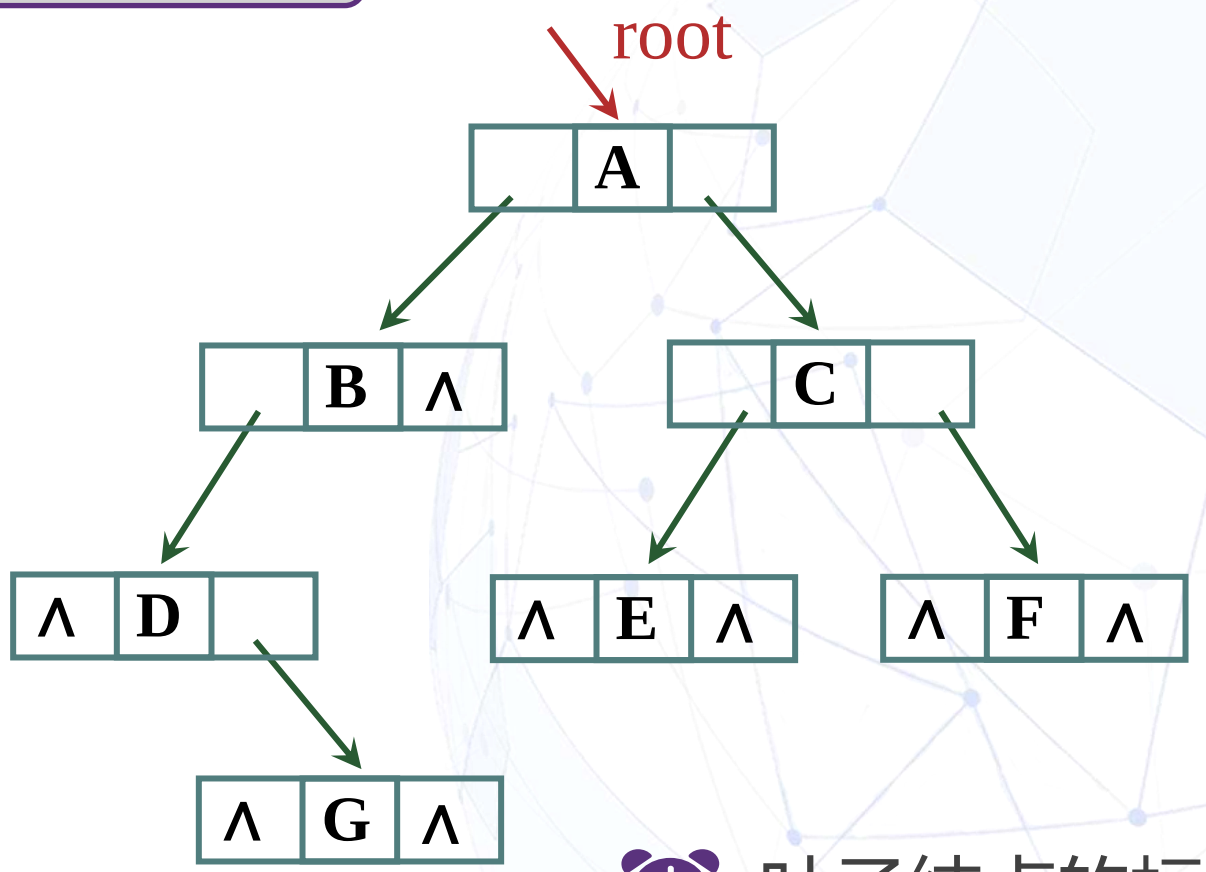


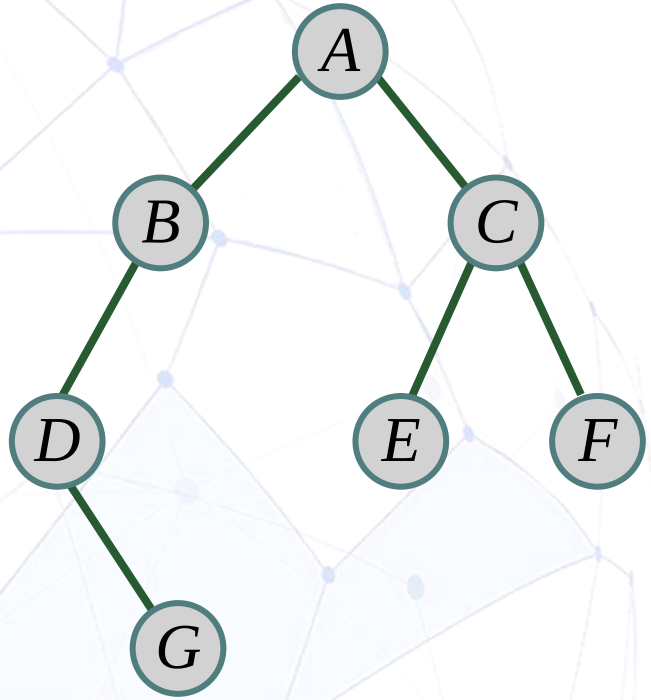
二叉链表：二叉树的每个结点对应一个链表结点，链表结点存放结点的数据信息和指示左右孩子的指针



lchild	data	rchild

二叉链表





叶子结点的标志？ ⇨ 左右孩子指针均为空

n 个结点的二叉链表有多少个空指针？

$2n-(n-1) = n+1$ 个

```
template <typename DataType>
struct BiNode{
    DataType data;
    BiNode *lchild, *rchild;
    BiNode(DataType val) {
        this->data = val; }
} ;
```

```
template <typename DataType>
class BiTree
{


……
    BiNode<DataType> *root;
};
```

单选题 5分

设置

1. 对于二叉树的顺序存储，是用一维数组按前序遍历存储结点。

A 正确

B 错误

提交

单选题 5分

设置

2. 二叉树的顺序存储一般仅用于存储完全二叉树。

A 正确

B 错误

提交

单选题 5分

设置

3. 在二叉链表中，叶子结点的左右孩子指针均为空指针。

A 正确

B 错误

提交

单选题 5分

设置

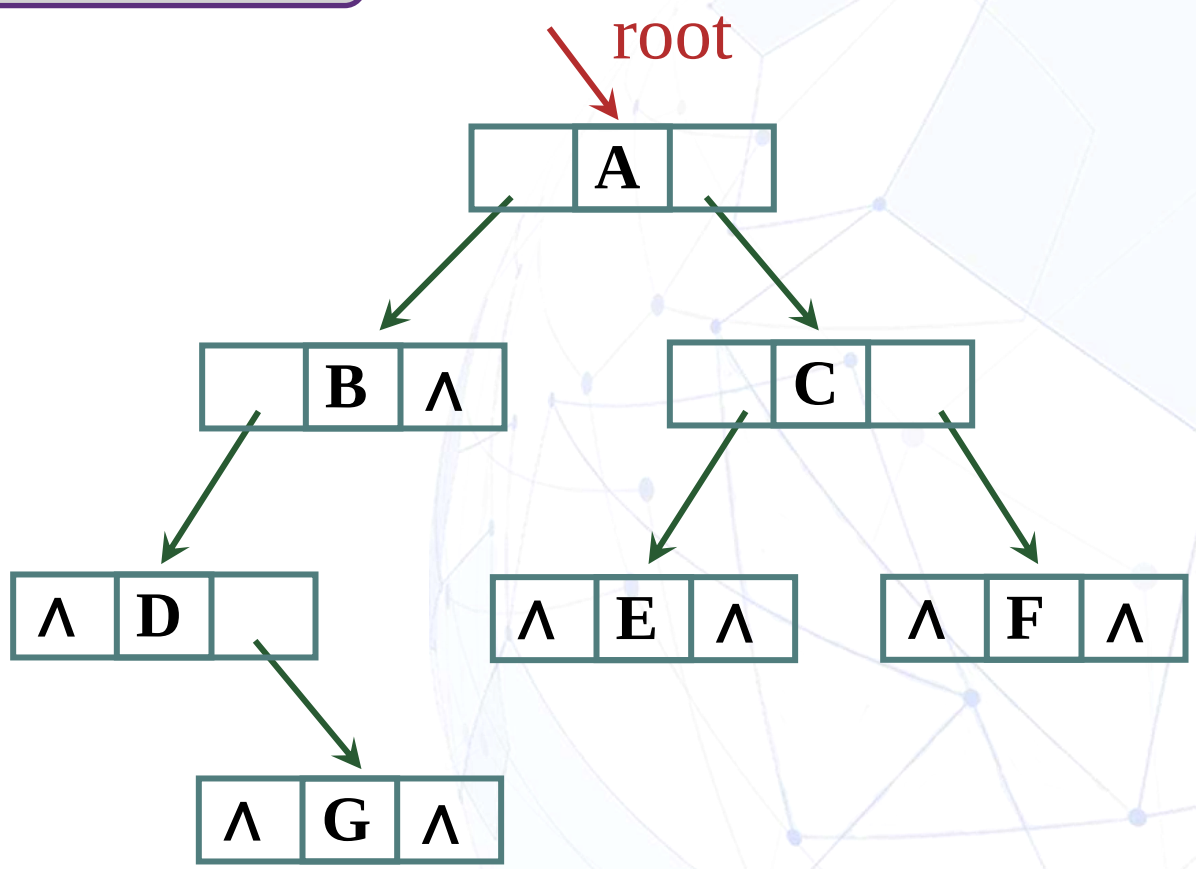
4. 在n个结点的二叉链表中有n+1个空指针，降低了空间利用率，因此，通常不用二叉链表存储二叉树。

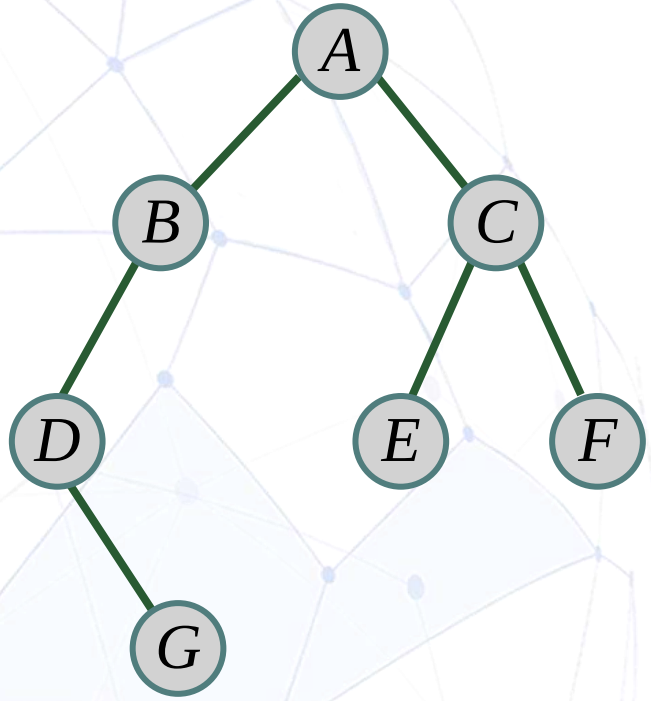
A 正确

B 错误

提交

二叉链表





lchild	data	rchild

如何查找双亲？时间性能？

单选题 5分

设置

5. 在二叉链表中查找某结点的双亲结点，平均情况下的时间复杂度是O(n)。

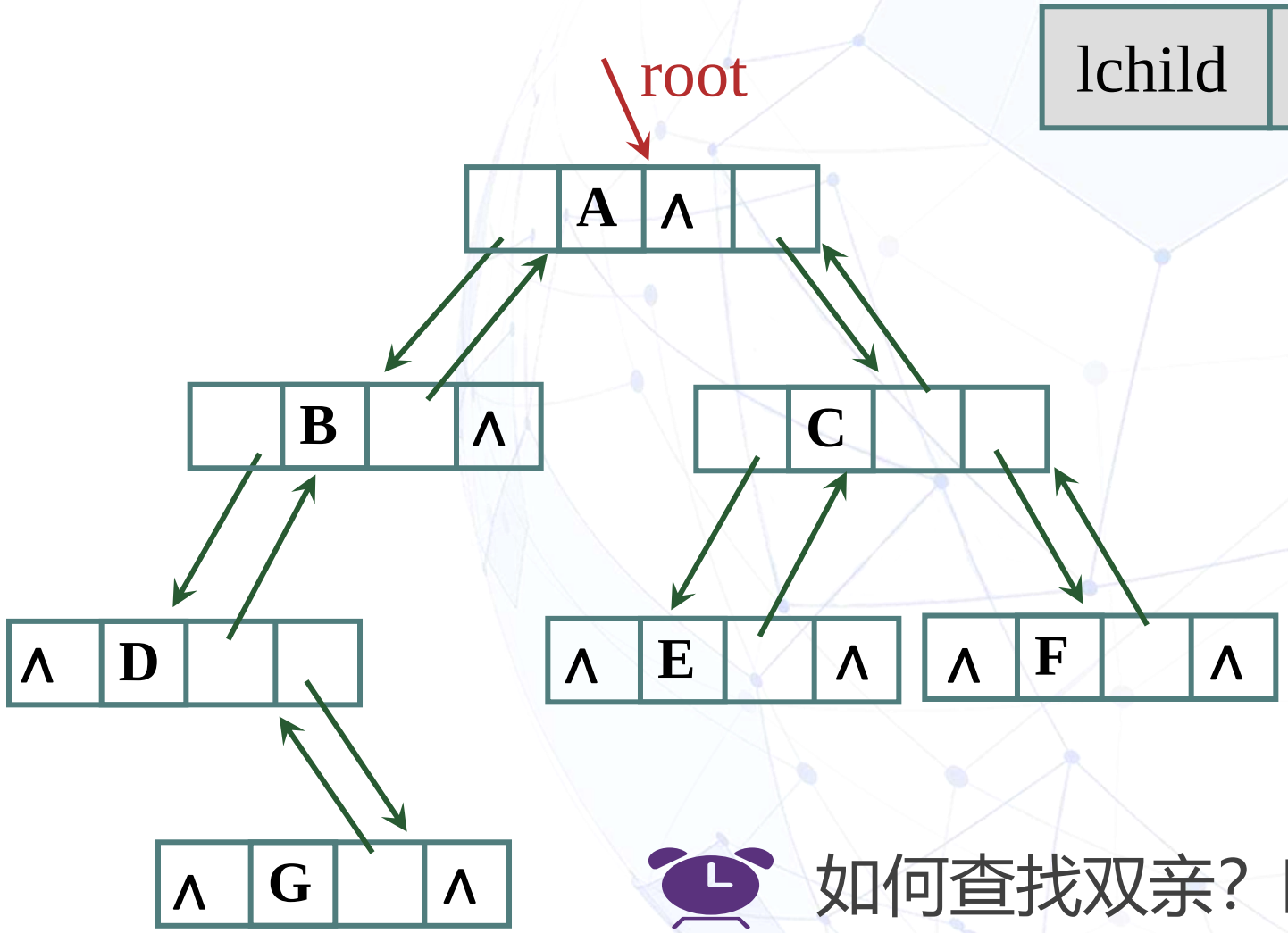
A 正确

B 错误

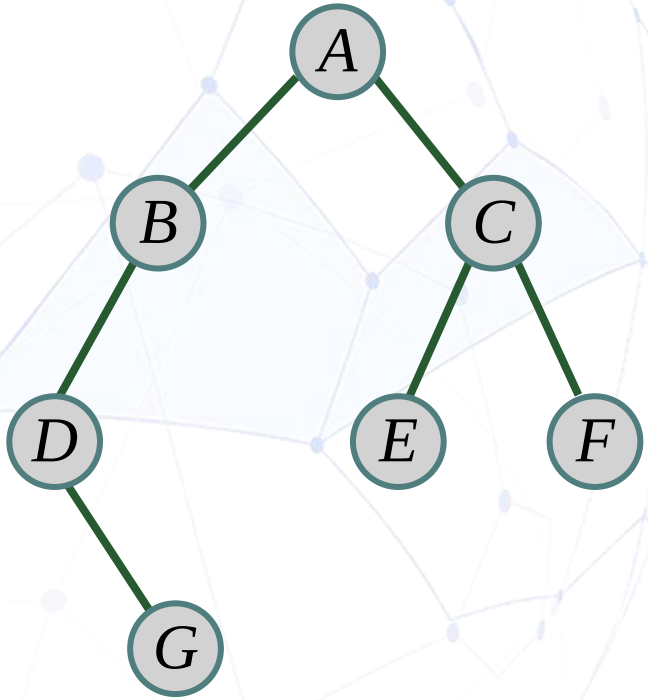
提交

三叉链表

在二叉链表中增加一个指向双亲的指针域



lchild	data	parent	rchild



如何查找双亲？时间性能？空间性能？

二叉链表

InitBiTree：初始化一棵空的二叉树

CreatBiTree：建立一棵二叉树

DestroyBiTree：销毁一棵二叉树

PreOrder：前序遍历二叉树

InOrder：中序遍历二叉树

PostOrder：后序遍历二叉树

LeverOrder：层序遍历二叉树

```
template <typename DataType>
struct BiNode{
    DataType data;
    BiNode *lchild, *rchild;
    BiNode(DataType val) {
        this->data = val; }
} ;
```

```
template <typename DataType>
class BiTree
{
public:
    BiTree( ){root = Creat( );}
    ~BiTree( ){Release(root);}
    void clear(){Release(root);}
    void PreOrder( ){PreOrder(root);}
    void InOrder( ){InOrder(root);}
    void PostOrder( ){PostOrder(root);}
    void LeverOrder( );
     ……
private:
    BiNode<DataType> *Creat( );
    void Release(BiNode<DataType> * &bt);
    void PreOrder(BiNode<DataType> *bt);
    void InOrder(BiNode<DataType> *bt);
    void PostOrder(BiNode<DataType> *bt);
   //safe guards
     ……
    BiNode<DataType> *root;
};
```

二叉树的遍历

二叉树的遍历：从根结点出发，按照某种次序访问树中所有结点，并且每个结点仅被访问一次

限定先左后右：前序、中序、后序

抽象操作，可以是对结点进行的各种处理，这里简化为输出结点的数据

按照什么次序对二叉树进行遍历呢？

二叉树 { 根结点D / 左子树L / 右子树R }

二叉树的遍历方式：

DLR、LDR、LRD、DRL、RDL、RLD

层序遍历：按二叉树的层序编号的次序访问各结点

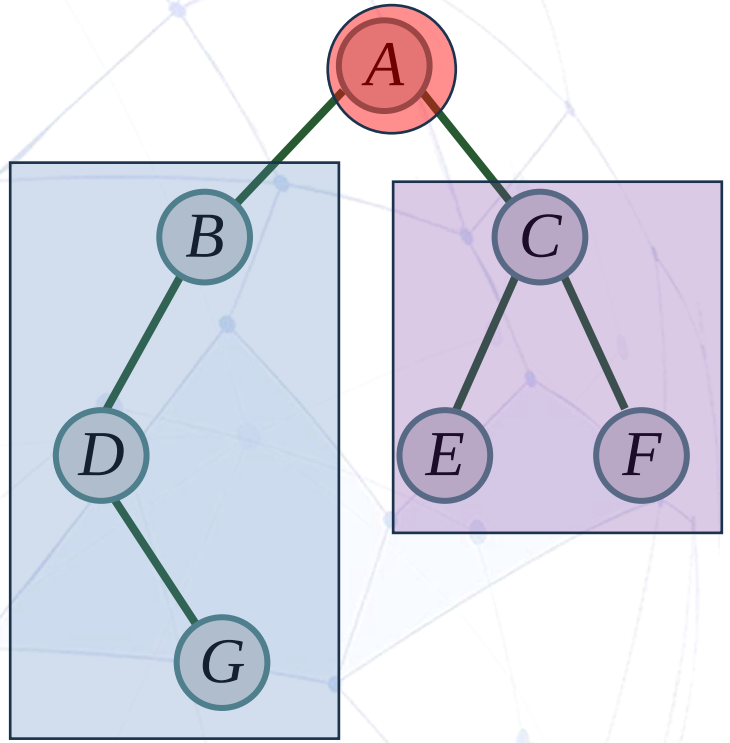
前序遍历

若二叉树为空，则空操作返回；否则：

（1）访问根结点

（2）前序遍历根结点的左子树

（3）前序遍历根结点的右子树



前序遍历序列：***A*** ***B D G*** ***C E F***

L R

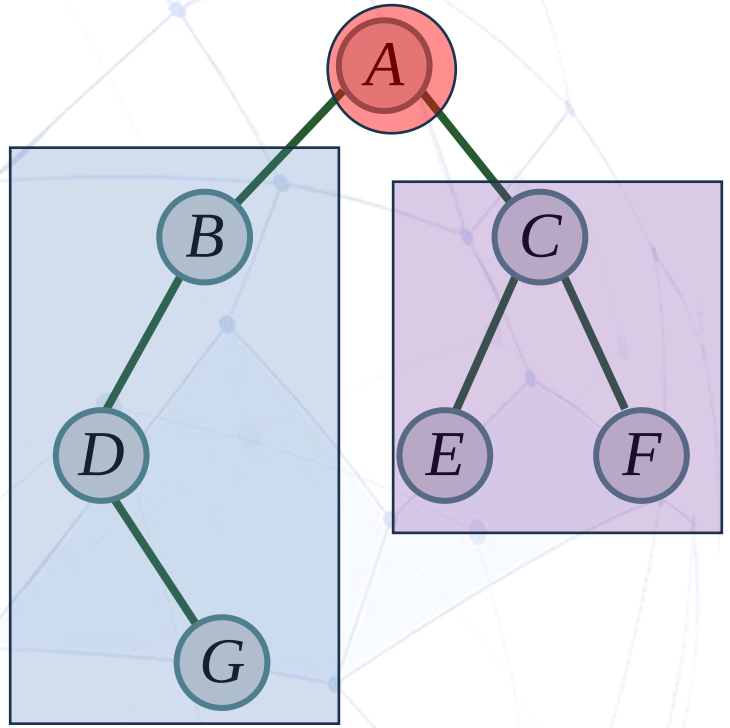
中序遍历

若二叉树为空，则空操作返回；否则：

（1）中序遍历根结点的左子树

（2）访问根结点

（3）中序遍历根结点的右子树



中序遍历序列：***D G B*** ***A*** ***E C F***

L R

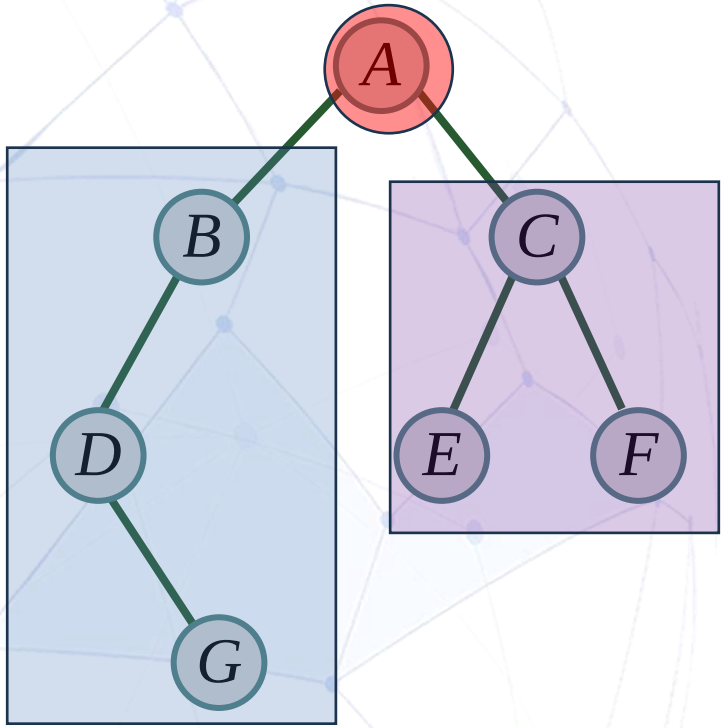
后序遍历

若二叉树为空，则空操作返回；否则：

（1）后序遍历根结点的左子树

（2）后序遍历根结点的右子树

（3）访问根结点

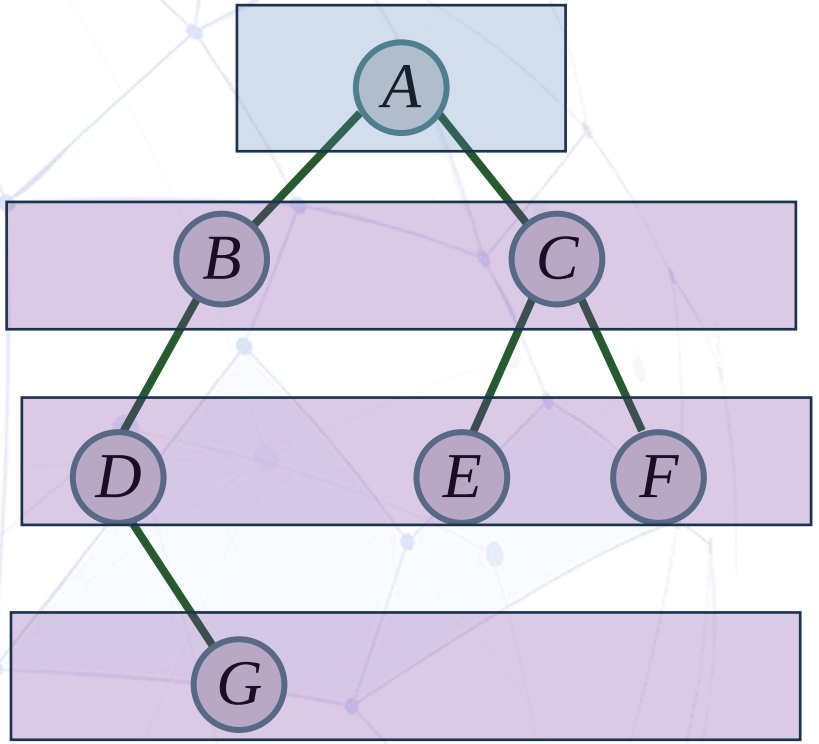


后序遍历序列：***G D B*** ***E F C*** ***A***

L R

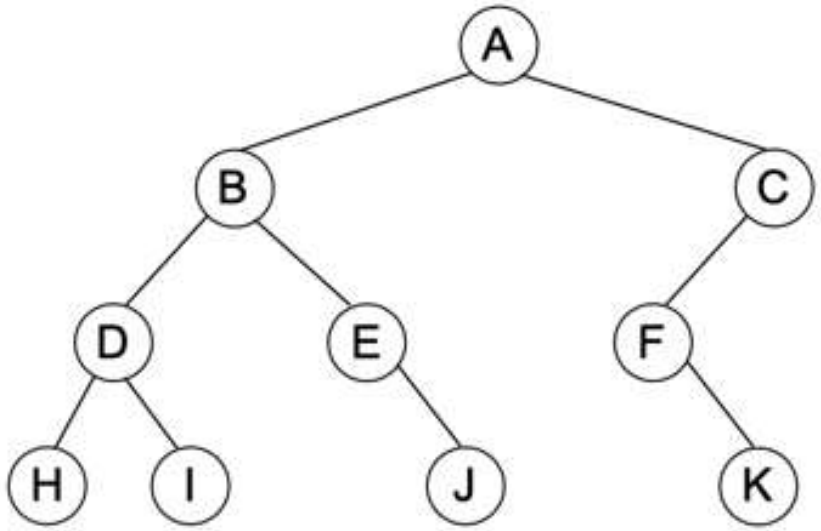
层序遍历

从二叉树的根结点开始，从上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问



层序遍历序列：*A B C D E F G*

练习1：遍历下列二叉树并按序输出结点数据

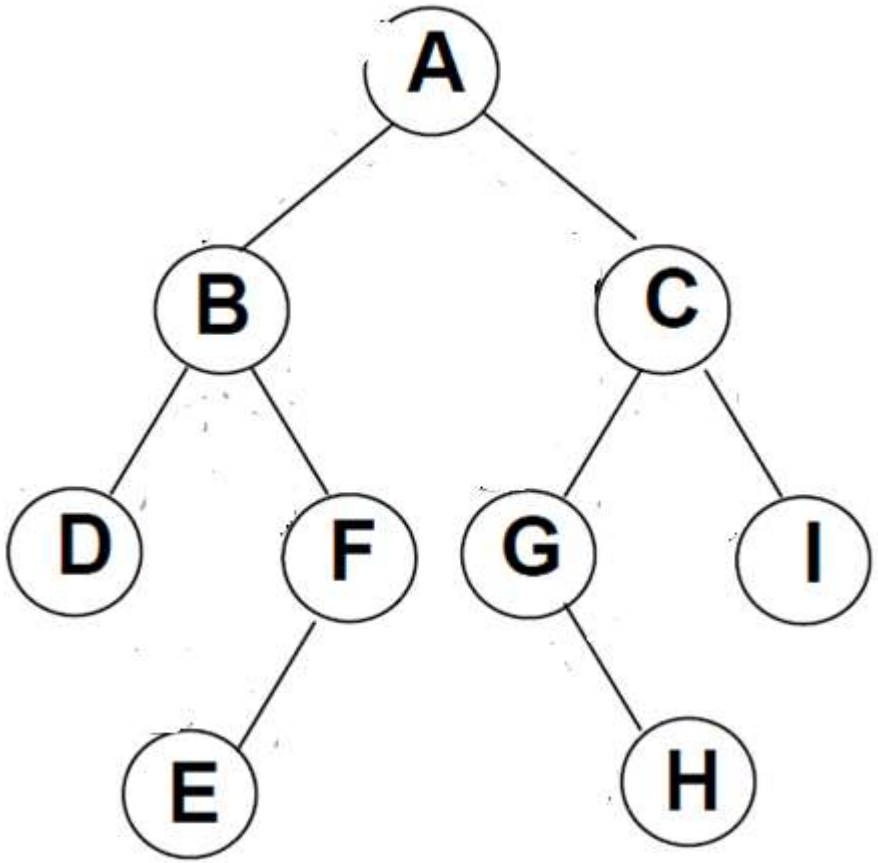


前序遍历：<A, B, D, H, I, E, J, C, F, K>

中序遍历：<H, D, I, B, E, J, A, F, K, C>

后序遍历：<H, I, D, J, E, B, K, F, C, A>

练习2



前序遍历序列:

中序遍历序列:

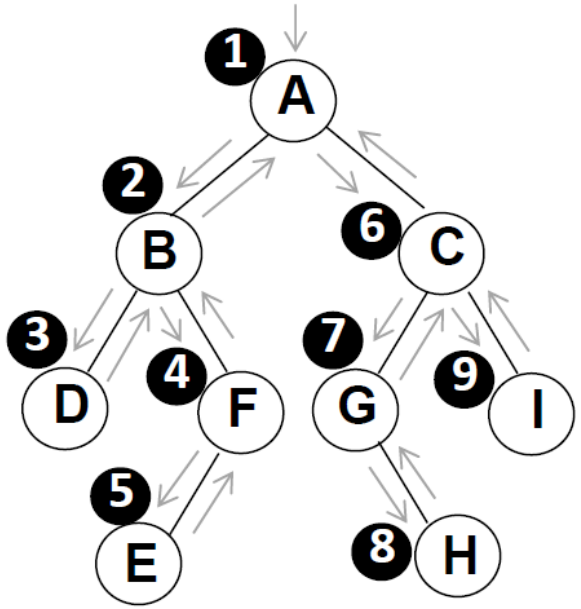
后序遍历序列:

A B D F E C G H I

D B E F A G H C I

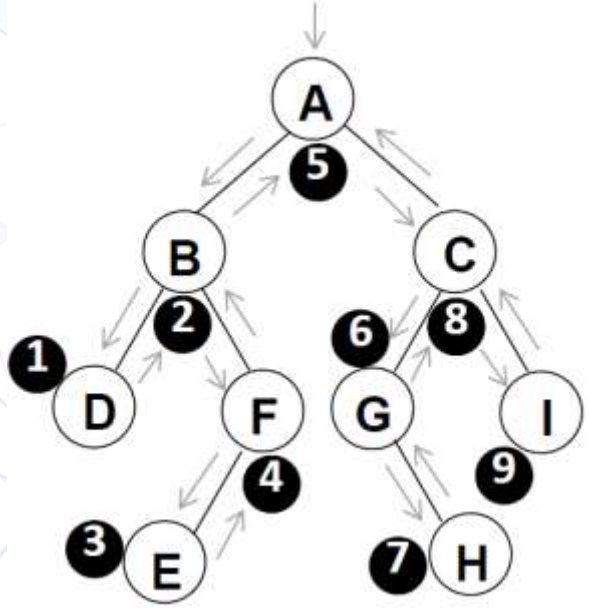
D E F B H G I C A

练习2



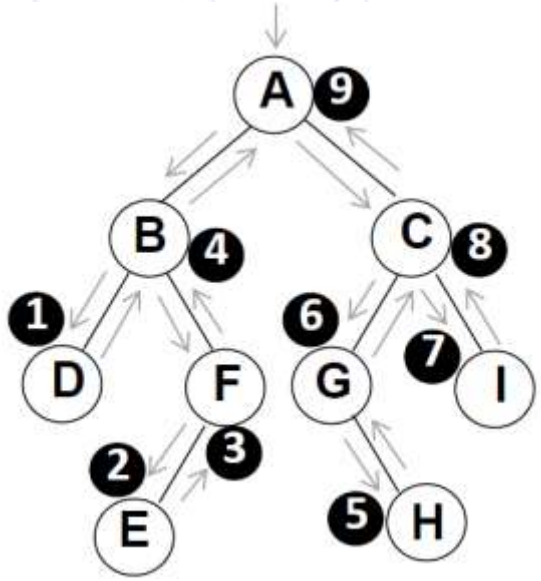
A（BDFE）（CGHI）

先序遍历=> A B D F E C G H I



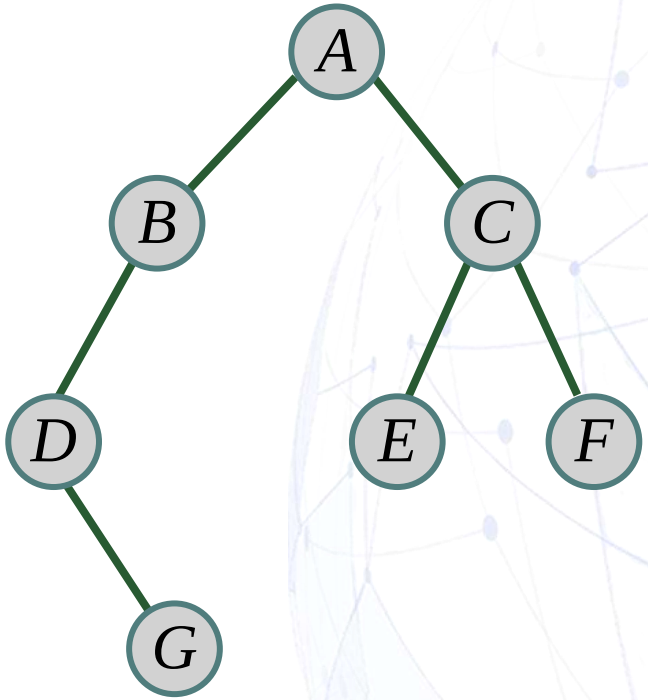
（D B E F）A（G H C I）

中序遍历=> D B E F A G H C I



（DEFB）（HGIC）A

后序遍历=> D E F B H G I C A



前序遍历序列：*A B D G C E F*

中序遍历序列：*D G B A E C F*

后序遍历序列：*G D B E F C A*

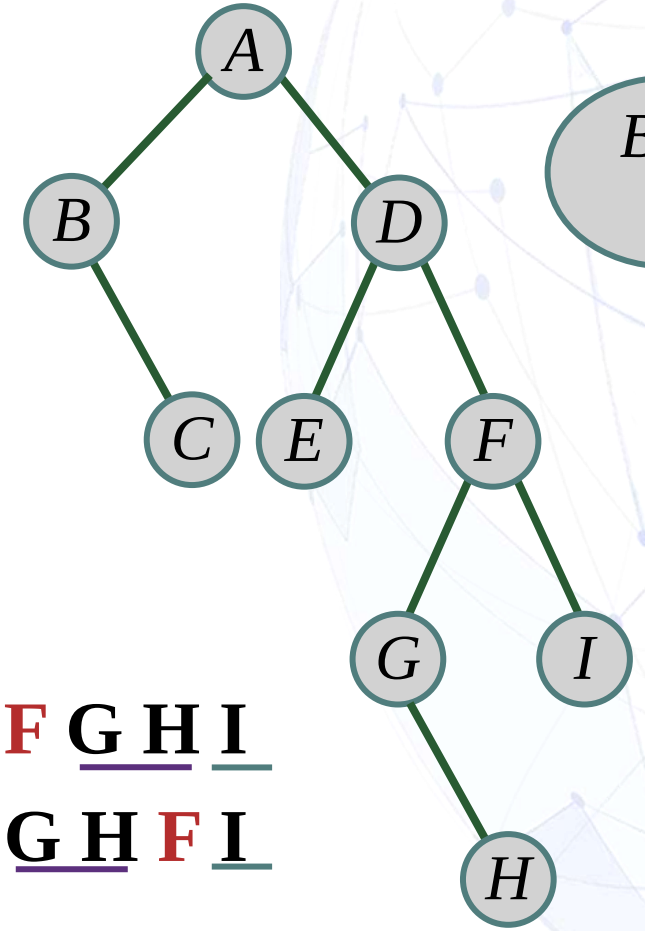
层序遍历序列：*A B C D E F G*

根结点位置：

- 前序遍历的最前
- 后序遍历的最后
- 中序遍历出现在左子树和右子树遍历结果之间

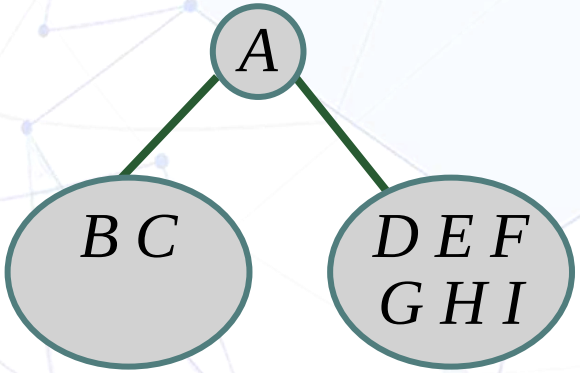
遍历与二叉树

若已知一棵二叉树的前序序列和中序序列，能否唯一确定这棵二叉树呢?



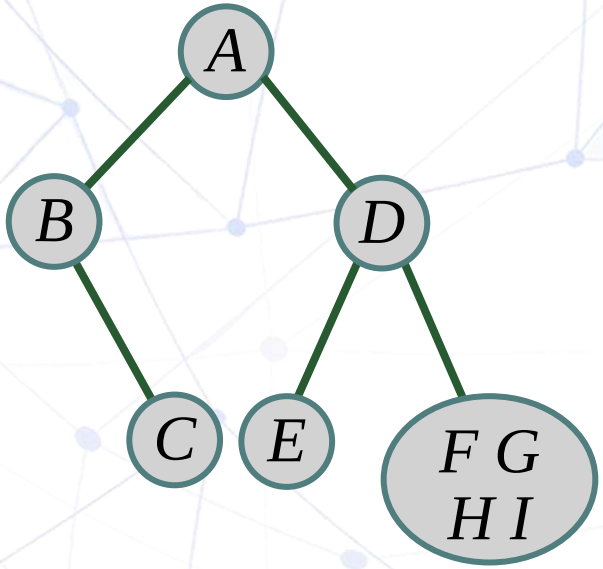
前序：F G H I

中序：G H F I



前序：A B C D E F G H I

中序：B C A E D G H F I



前序：B C

中序：B C

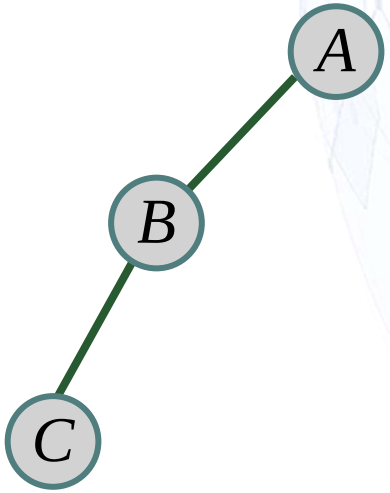
前序：D E F G H I

中序：E D G H F I

遍历与二叉树

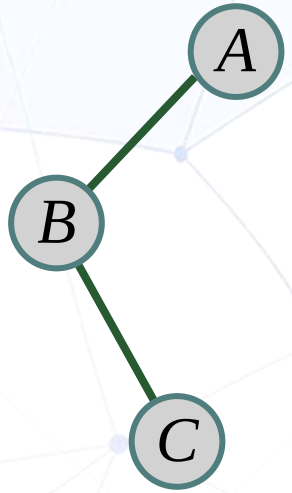
- 已知一棵二叉树的前序（或中序，或后序，或层序）序列，**不能**唯一确定这棵二叉树（特殊处理后，比如扩充二叉树后可以）。

- 若已知一棵二叉树的前序序列和后序序列，能否唯一确定这棵二叉树呢?



前序遍历序列：***A B C***

后序遍历序列：***C B A***



基于二叉链表的二叉树遍历算法

也称为**序列化**算法

- 深度遍历递归算法：前序、中序、后序遍历
- 广度(层次)遍历算法
- 深度遍历非递归算法（选学）

深度遍历递归算法

算法5-2. 前序遍历二叉树 PreOrder(*tree*)

if tree ≠ NIL **then** //空树不做处理，直接返回
| Visit(*tree*) //访问根结点
| PreOrder(*tree.left*) //前序遍历左子树
| PreOrder(*tree.right*) //前序遍历右子树
end

算法5-3. 中序遍历二叉树 InOrder(*tree*)

if *tree* ≠ NIL **then**
| Inorder(*tree.left*) //中序遍历左子树
| Visit(*tree*) //访问根结点
| InOrder(*tree.right*) //中序遍历右子树
end

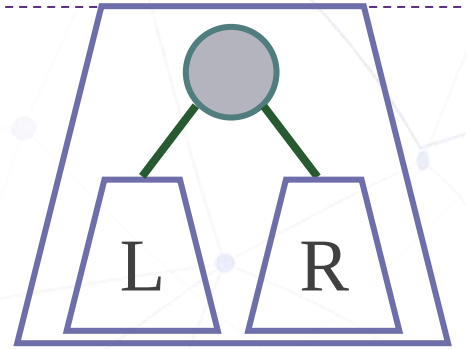
算法5-4. 后序遍历二叉树 PostOrder(*tree*)

if *tree* ≠ NIL **then**
| Postorder(*tree.left*) //后序遍历左子树
| PostOrder(*tree.right*) //后序遍历右子树
| Visit(*tree*) //访问根结点
end

前序遍历

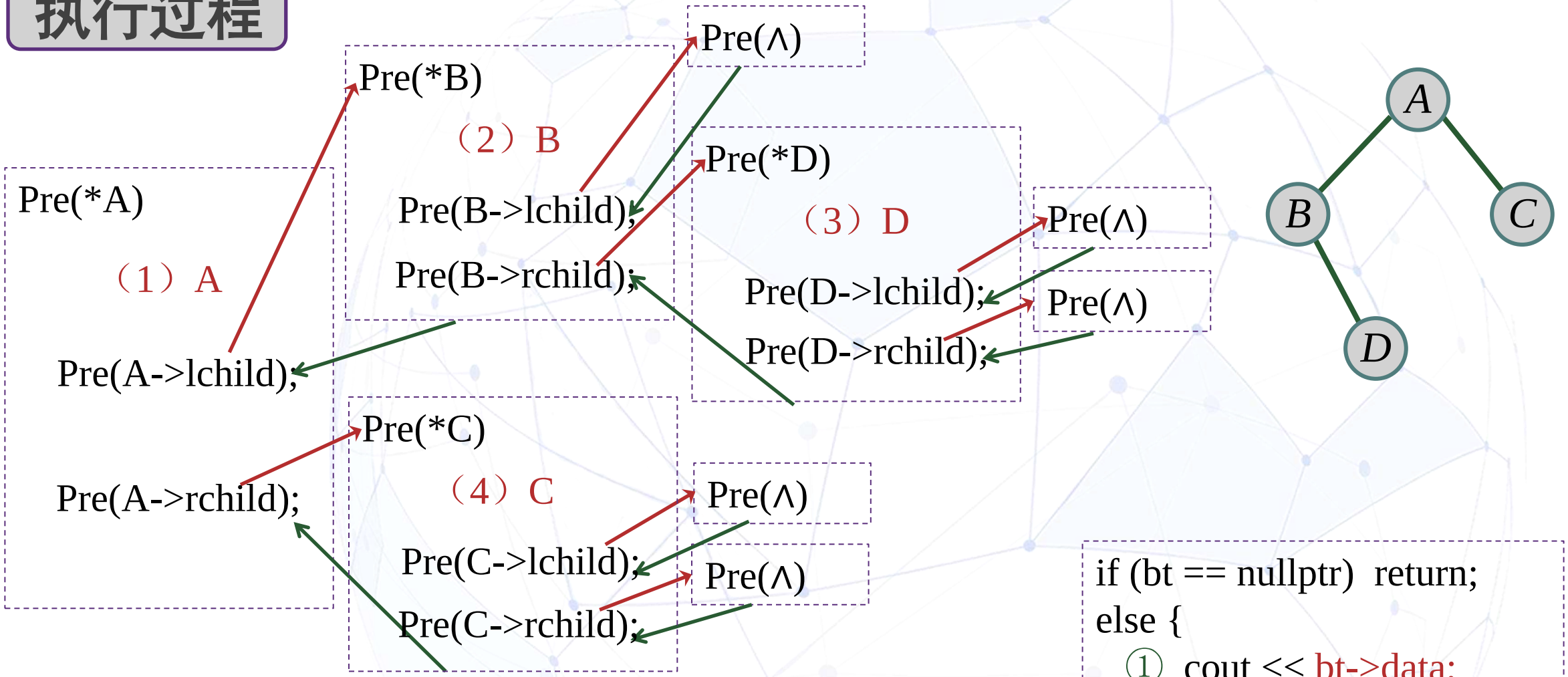
```
template <typename DataType>
void BiTree<DataType> :: PreOrder(BiNode<DataType> *bt)
{
    if (bt == nullptr) return;                    //递归调用的结束条件
    else {
        cout << bt->data;                         //访问根结点bt的数据域
        PreOrder(bt->lchild);                     //前序递归遍历bt的左子树
        PreOrder(bt->rchild);                     //前序递归遍历bt的右子树
    }
}
```

按照先左后右的方式扫描二叉树，
区别仅在于访问结点的时机



中序遍历、后序遍历算法?

执行过程

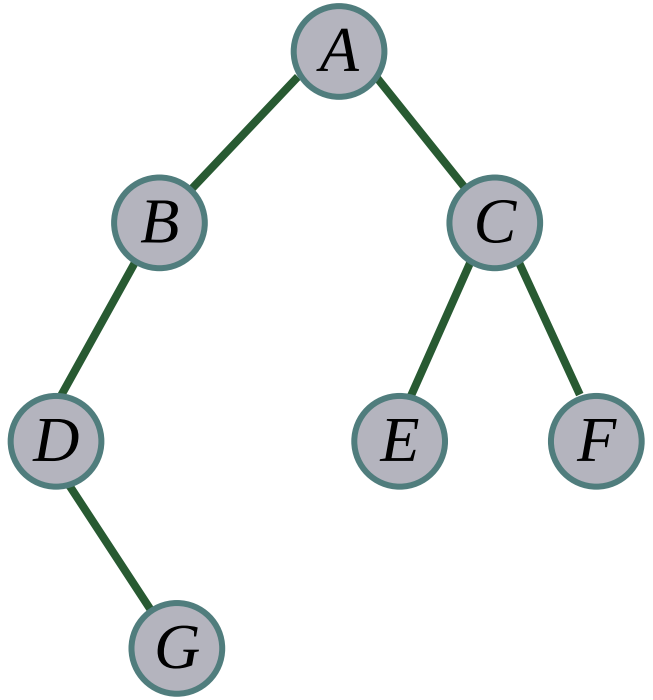



```
if (bt == nullptr)  return;
else {
   ①  cout << bt->data;
   ②  PreOrder(bt->lchild);
   ③  PreOrder(bt->rchild);
}
```

约定：*A 表示根指针指向结点 A

二叉树的层序遍历（广度遍历）算法

1. 队列 Q 初始化；
2. 如果二叉树非空，将根指针入队；
3. 循环直到队列 Q 为空
   3.1 q = 队列 Q 的队头元素出队；
   3.2 访问结点 q 的数据域；
   3.3 若结点 q 存在左孩子，则将左孩子指针入队；
   3.4 若结点 q 存在右孩子，则将右孩子指针入队；



出队 ← A B C D E F G ← 入队

遍历序列：A B C D E F G

层序遍历算法实现

```
template <typename DataType>
void BiTree<DataType> :: LeverOrder( )
{
    BiNode<DataType> *Q[100], *q = nullptr;
    int front = −1, rear = −  1;
    if (root == nullptr) return;
    Q[++rear] = root;
    while (front != rear)
    {
        q = Q[++front];      cout << q->data;
        if (q->lchild != nullptr)  Q[++rear] = q->lchild;
        if (q->rchild != nullptr)  Q[++rear] = q->rchild;
    }
}
```

$O(n)$

时间复杂度? ⇨ 每个结点进队出队一次 ⇨ $O(n)$

思考：

对二叉树的结点从1开始进行连续编号，要求每个结点的编号大于其左、右孩子的编号，同一结点的左、右孩子中，其左孩子的编号小于其右孩子的编号，是采用何种次序的遍历实现编号的。

二叉树遍历的非递归算法

递归算法的问题：

(1) 存在不支持递归算法的程序设计语言

(2) 递归算法在运行中，需要系统在内存栈中分配空间保存函数的参数、返回地址以及局部变量等，运行效率较低

(3)系统对每个进程分配的栈容量有限，如果二叉树的深度太大造成递归调用的层次太高，容易导致栈溢出

非递归算法实现的关键：使用栈结构模拟函数调用中系统栈的工作原理

以中序遍历为例 

中序遍历操作的非递归实现

二叉树中序遍历的非递归算法的关键：在中序遍历过某结点的整个左子树后，如何找到该结点的根以及右子树。

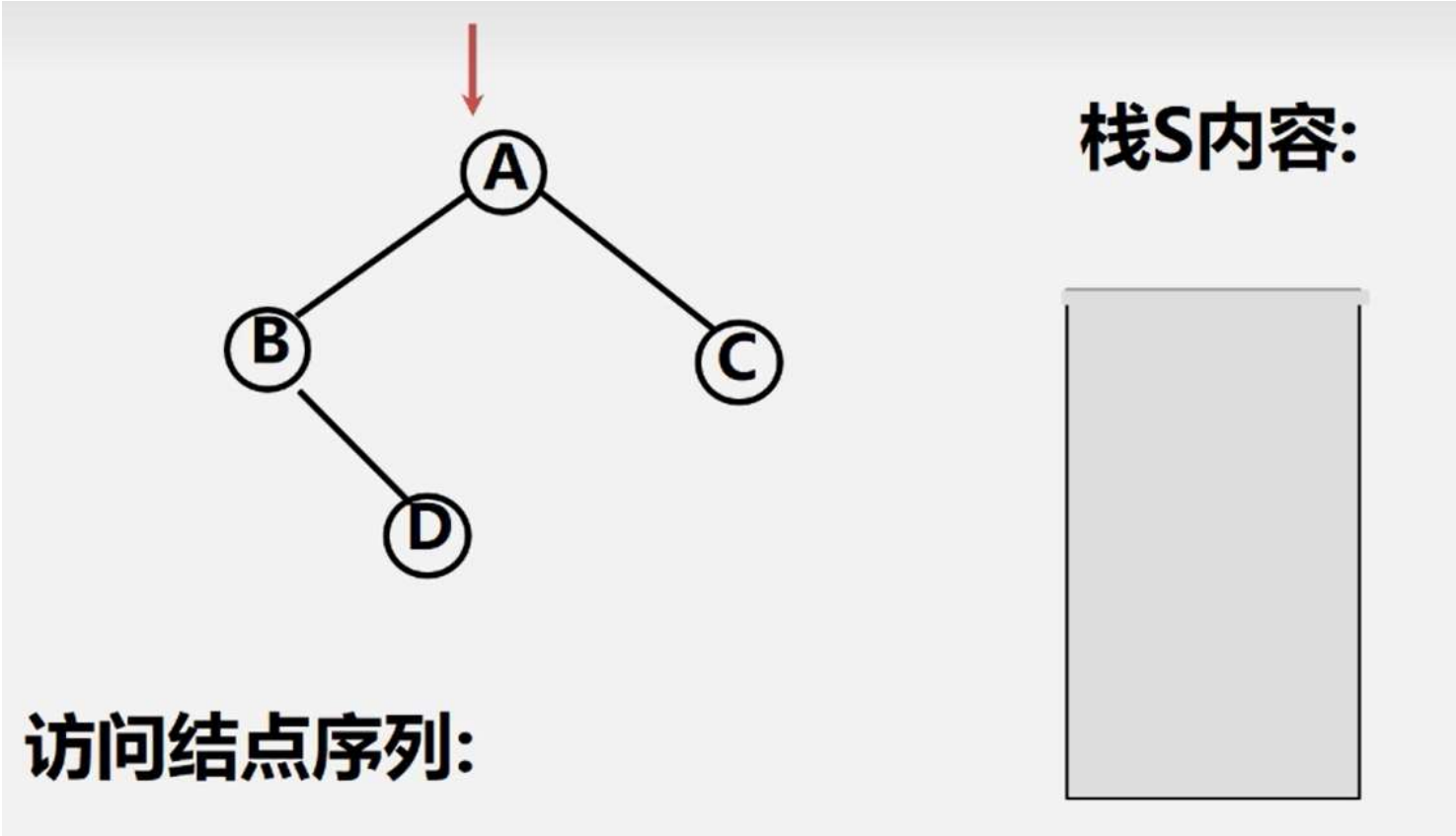
基本思想：

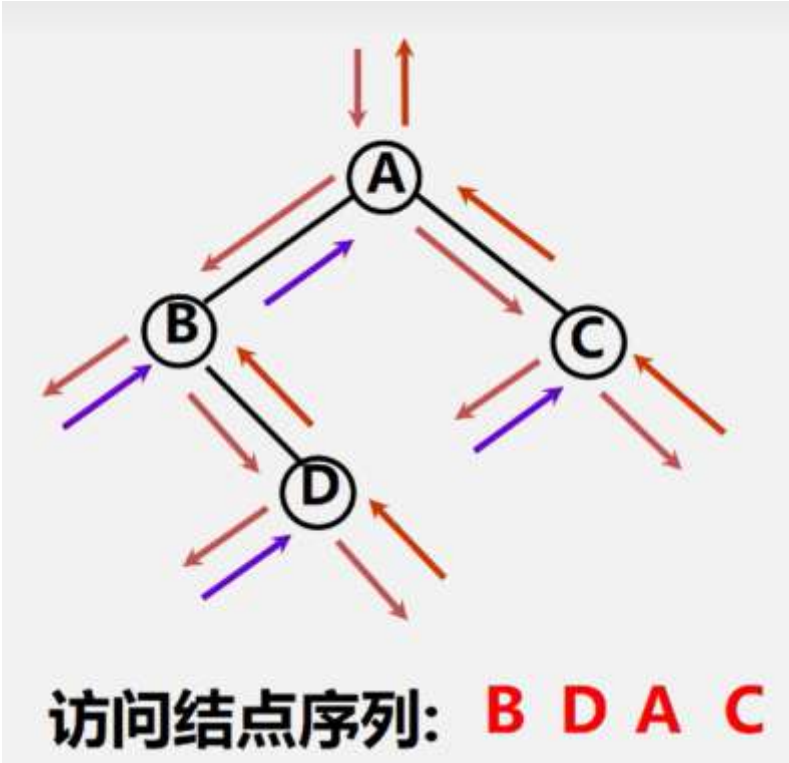
（1）建立一个栈

（2）根结点进栈，遍历左子树

（3）根结点出栈，输出根结点，遍历右子树。

中序遍历操作的非递归实现





沿左分支下移，并将经过的结点压入栈

左子树为空，或者左子树遍历结束，则从栈中弹出结点，开始遍历结点的右子树

中序遍历的非递归实现

1. 初始化一个空栈，设为s；
2. 设活动指针p将在二叉树各结点中移动，初始指向root；
3. 当栈s不为空或p不为空时，循环执行：

   A. 当p不为空时，循环执行：p入栈于s，p往其左孩子移动；

   B. 如栈非空，则出栈s中一个结点，假设也为p，访问p所指结点，p调整至其右孩子，如右孩子为空，则继续出栈（执行B），否则，执行A，这个步骤可统一成继续循环执行步骤3，直至p为空且栈s也为空。

算法5-8：非递归中序遍历 InOrder(*tree*)

```
InitStack(stack)    //初始化栈stack，用于存放结点
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do        //当前结点不是空结点
| | Push(stack, tree)   //结点压入栈
| | tree ← tree.left     //沿左分支下移
| end
| if IsEmpty(stack) = false then   //如果栈不为空
| | tree ← Top(stack)
| | Visit(tree)          //访问栈顶结点
| | Pop(stack)           //弹出栈顶结点
| | tree ← tree.right  //移到栈顶结点的右子树
| end
end
DestroyStack(stack)
```

沿左分支下移，并将经过的结点压入栈

tree=NIL表示左子树为空，或者左子树遍历结束，则从栈中弹出结点，开始遍历结点的右子树

中序遍历操作的非递归实现（c++）

```
#include "seqstack.h"
template <class DataType>
void BiTree<DataType>::inorder() {
        if (root != nullptr) {
                SeqStack<BiNode<DataType> *> s;
                BiNode<DataType>*p = root;
                while (!s.Empty() || p != nullptr) {
                        while (p != nullptr) {
                                s.Push(p); p = p->lchild;
                        }
                        if (!s.Empty()) {
                                p=s.Pop();
                                cout<<p->data;
                                p = p->rchild;
                        }
}
}
}
```

前序遍历的非递归实现（自学）

算法的关键：在前序遍历过某结点的整个左子树后，如何找到该结点的右子树的根指针

1、 初始化一个空栈，设为s；

2、 设活动指针p将在二叉树各结点中移动，初始指向root；

3、 当栈s不为空或p不为空时，循环执行：

A. **当p不为空时，循环执行**：p入栈于s，访问p所指结点，p往其左孩子移动；

B. 如栈非空，则出栈s中一个结点，假设也为p，p调整至其右孩子，如右孩子为空，则继续出栈（执行B），否则，执行A，这个步骤可统一成继续循环执行步骤3，直至p为空且栈s也为空。

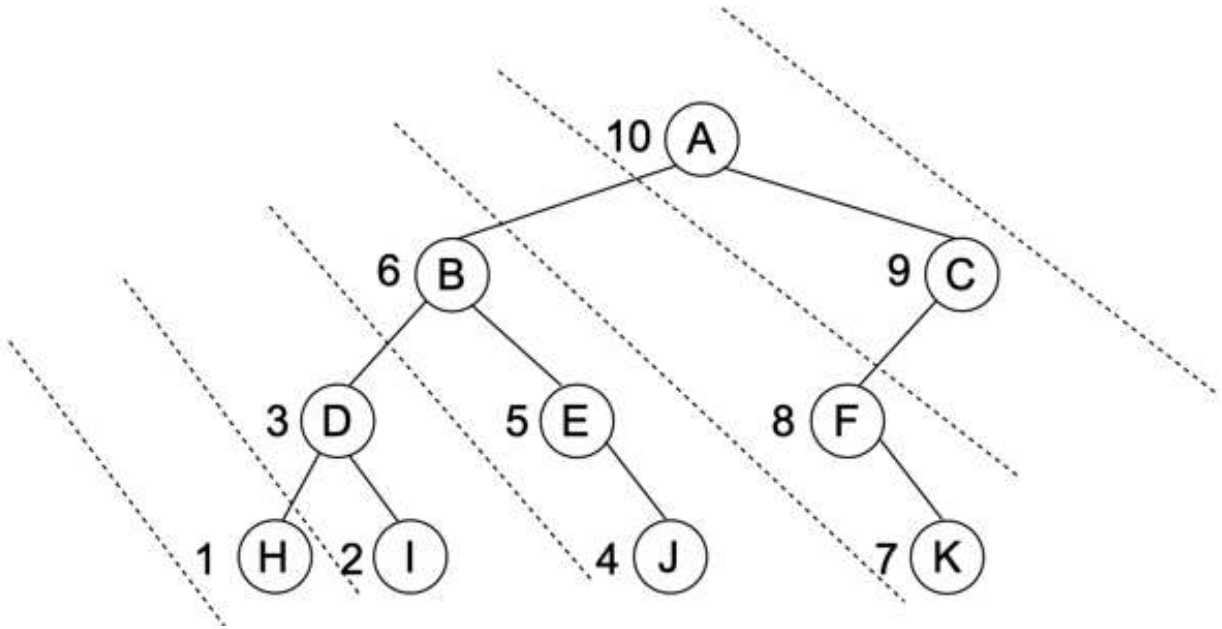
算法5-7：非递归前序遍历 PreOrder(*tree*)

```
InitStack(stack)    //初始化栈stack，用于存放结点
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do        //当前结点不是空结点
| | Visit(tree)              //访问结点
| | Push(stack, tree)        //结点压入栈
| | tree ← tree.left         //沿左分支下移
| end
| if IsEmpty(stack) = false then   //如果栈不为空
| | tree ← Top(stack)
| | Pop(stack)               //弹出栈顶结点
| | tree ← tree.right        //移到栈顶结点的右子树
| end
end
DestroyStack(stack)
```

沿左分支下移，并将经过的结点压入栈

tree=NIL表示左子树为空，或者左子树遍历结束，则从栈中弹出结点，开始遍历结点的右子树

后序遍历算法的非递归化（自学）



二叉树所有结点的后序遍历次序，用各结点左边的数字表示

后序遍历次序的结点序列可分为多个段，用虚线分开，每段中的结点具有以下性质：

(1) 段中各结点的访问次序是连续的，并且最先访问（次序最小）的结点没有右子结点

(2) 段中若有多个结点，则次序相邻的任意两个结点，次序小的结点是次序大的结点的右子结点

(3) 段中次序最大的结点如果不是根，则是其父结点的左子结点

算法5-9：非递归后序遍历 PostOrder(*tree*)

```
InitStack(stack)
while tree ≠ NIL 或 IsEmpty(stack) = false do
|  while tree ≠ NIL do  //当前结点不是空结点
|  |  Push(stack, tree)   //结点压入栈
|  |  tree ← tree.left  //沿左分支下移
|  end
|
|  top ← Top(stack)     //stack非空，top指向栈顶结点
|  pre_top ← NIL        //初始化 pre_top
|                       //如果栈顶结点的右子树为空，开始从栈弹出结点
```

沿左分支下移，
并将经过的结点
压入栈

开始时，*pre_top* = NIL

(continue)

```
| while IsEmpty(stack) = false 且 top.right = pre_top  do
| | Visit(top)          //访问当前栈顶结点
| | pre_top ← top     //栈顶结点传给pre_top
| | Pop(stack)          //弹出栈顶结点
| | if IsEmpty(stack) = false then
| | | top ← Top(stack)   //栈非空，top指向新的栈顶结点
| | else
| | | top ← NIL      //空栈，top赋值NIL，结束遍历
| | end
| end
| if top ≠ NIL then
| | tree ← top.right   //移到栈顶结点的右子树并开始遍历
| end
end
DestroyStack(stack)
```

从右子树为空的结点开始。依次弹出各段中的结点

若弹出的结点是新栈顶结点的右子结点，继续弹出

二叉树的反序列化

即根据遍历序列构造二叉树

二叉树的序列化：

按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

序列化的作用：将树的非线性结构转换成线性结构，便于使用线性表或字符串等存储

二叉树的反序列化：

根据线性序列重构原始的二叉树，即**二叉树的构造**

二叉树的反序列化

二叉树的反序列化：根据线性序列重构原始的二叉树

- 完全二叉树的顺序存储是一种序列化方案，并且可以根据结点间的相对位置确定它们之间的逻辑关系，重构出二叉树
- 但该方法对于一般的二叉树可能造成空间浪费，在最坏情况下，空间复杂度达到$O(2^n)$

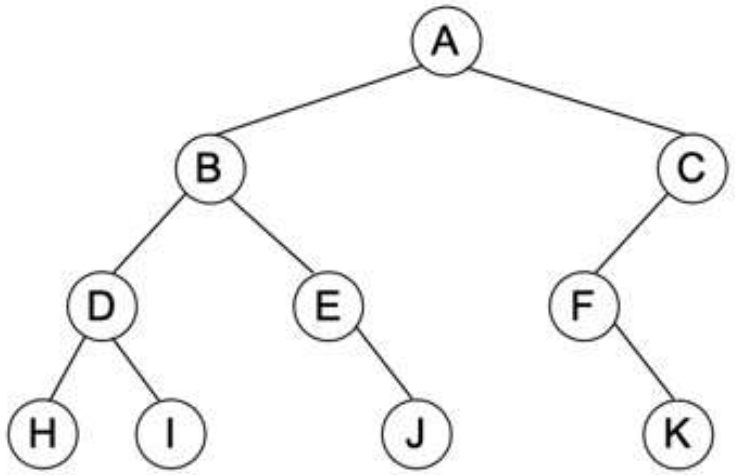
- 常用的前序遍历或层序遍历算法，产生的结点序列只包含了树结构的部分信息，通常无法重构二叉树

方法1-基于扩充二叉树的前序序列化与反序列化

（1）基于扩充二叉树的前序序列化

（2）基于扩充前序序列的反序列化

方法1-基于扩充二叉树的前序序列化与反序列化



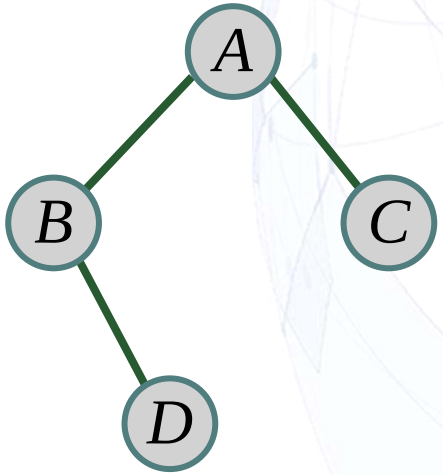
前序遍历：<A, B, D, H, I, E, J, C, F, K>

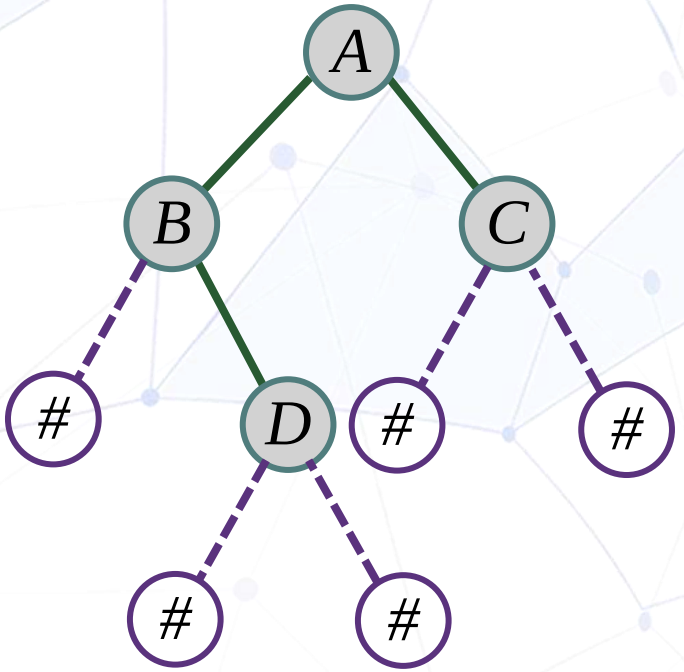
从序列中，最多只能确定A是根结点，其它信息，如左子树包含哪些结点等无法确定

问题：如何使前序遍历的结果能够重构二叉树？

(1) 基于扩充二叉树的前序序列化

扩充（展）二叉树：将二叉树中每个结点的空指针引出一个虚结点，其值为一特定值如 '#'





扩充二叉树的前序遍历序列：**A B # D # # C # #**

通过在结点序列中插入空记号，记录二叉树的非线性结构

算法5-11：扩充的二叉树前序序列化PreOrderSerialize(*tree*)

输入：二叉树*tree*
输出：二叉树的前序序列

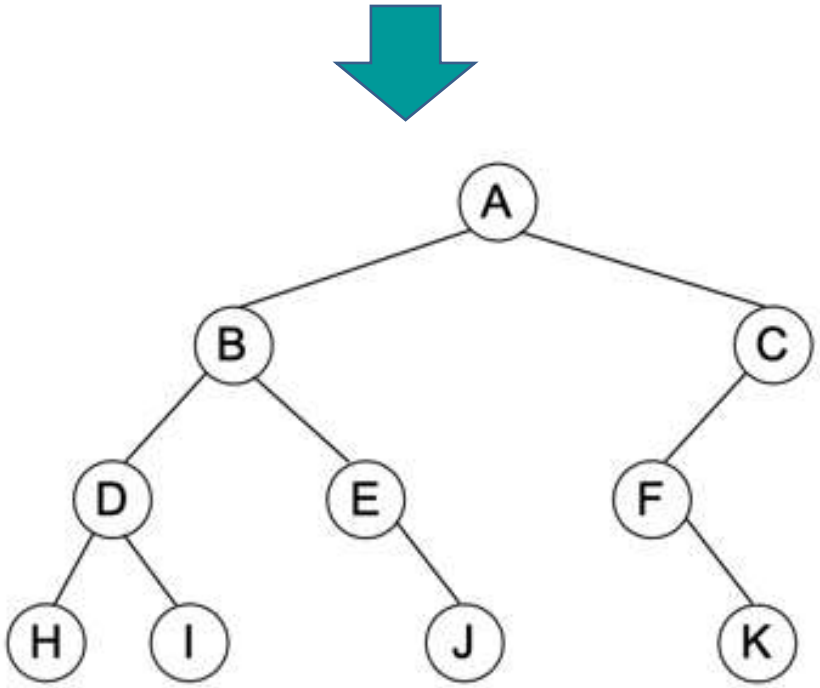
```
if tree = NIL then  //空树
|  print  #     //输出特殊符号，代表空结点
else
|  print  tree.data   //输出结点数据
|  PreOrderSerialize (tree.left)      //对左子树前序序列化
|  PreOrderSerialize (tree.right)     //对右子树前序序列化
end
```

基于前序遍历算法

（2）基于扩充前序序列的反序列化

已知一个扩充前序序列，根据该序列构建二叉树

<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>



从前序序列先端依次读取数据，执行下面的操作：

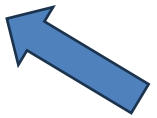
- 如果读取的数据是#，则返回NIL，表示空结点或空树
- 否则新建二叉树结点，把数据代入结点并递归地重构结点的左子树和右子树，然后返回结点。

算法5-12：基于扩充前序序列的反序列化 PreOrderDeSerialize(*preorder*, *n*)

输入：存放二叉树前序序列的线性表*preorder*，表中元素个数*n* (n>0)

输出：二叉树

全局变量：k，初始值为-1



此处前序序列是一个静态字符串

```
k ← k + 1
tree ← NIL  //初始化一个空树
if k < n then  //k是线性表的有效序号
| data ← Get(preorder, k)       //读出线性表第k个元素
| if data ≠ # then      //非空记号
| | tree ← new BinaryTreeNode( )  //新建二叉树结点
| | tree.data ← data              //代入数据
| | tree.left ← PreOrderDeSerialize(preorder, n)  //重构左子树
| | tree.right ← PreOrderDeSerialize(preorder, n)  //重构右子树
| end
end
return tree  //返回新建的二叉树或空树
```

基于扩充前序序列的二叉树构造

```
template <typename DataType>
BiNode<DataType> *BiTree<DataType> :: Creat( )
{
    char ch;
    cin >> ch;                                  //输入结点的数据信息，假设为字符
    if (ch == '#') bt = nullptr;                //建立一棵空树
    else {
        bt = new BiNode<DataType>(ch);
        bt->lchild = Creat( );                  //递归建立左子树
        bt->rchild = Creat( );                  //递归建立右子树
    }
    return bt;
}
```

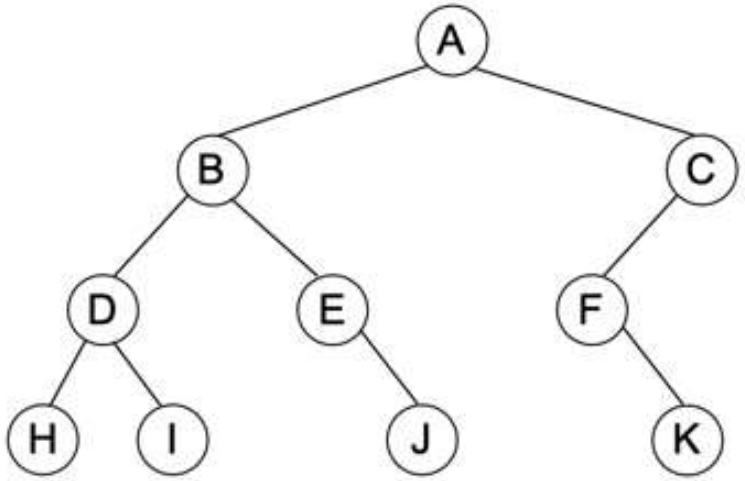
与前面算法不同，此处前序序列是通过动态输入的

方法（2）-基于二叉树前序和中序序列的反序列化

(经典问题）用前序遍历与中序遍历结果重构二叉树

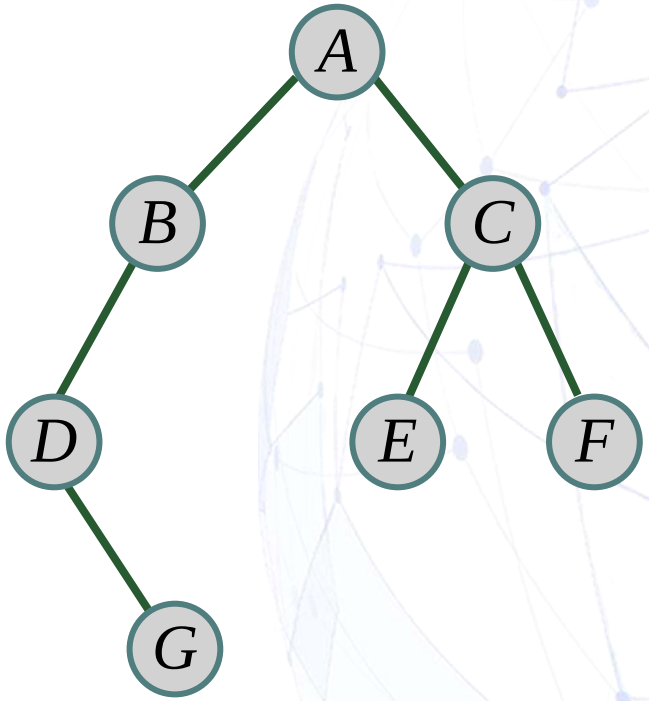
前序遍历：<A, B, D, H, I, E, J, C, F, K>

中序遍历：<H, D, I, B, E, J, A, F, K, C>



```
template <class DataType>
BiNode<DataType> * BiTree<DataType>::creat2(string preorder, string inorder){
        if (preorder.length() == 0)
                return nullptr;
        char rootchar = preorder[0];
        int rootin = inorder.find(rootchar);
        string leftin = inorder.substr(0,rootin);
        string rightin = inorder.substr(rootin+1);
        string leftpre = preorder.substr(1,rootin );
        string rightpre = preorder.substr(rootin + 1);
        BiNode<DataType> *bt =new  BiNode<DataType>(rootchar);
         bt->lchild = creat2(leftpre, leftin);
         bt->rchild = creat2(rightpre, rightin);
        return bt;
}
```

思考：二叉树的层次序列化和反序列化(自学)



层序遍历序列：***A B C D E F G***

扩充的层序遍历序列：

A B C D # E F # G # # # # # #

反序列化的核心思想：读取序列第一个数据并创建根结点，把根节点插入队列。接下来每次从队首弹出一个结点，并从层序序列中连续取出2个数据，直到完序列中所有数据。详见P61算法5-13和5-14

析构函数

为什么要销毁内存中的二叉链表?

二叉链表是动态存储分配，二叉链表的结点是在程序运行过程中动态申请的，在二叉链表变量退出作用域前，要释放二叉链表的存储空间

```
template <typename DataType>
void BiTree<DataType> :: Release(BiNode<DataType> * &bt)
{
    if (bt == nullptr) return;
    else{
        Release(bt->lchild);          //释放左子树
        Release(bt->rchild);          //释放右子树
        delete bt;                    //释放根结点
         bt=nullptr;
    }
}
```

除遍历外的其它二叉树操作（链表结构）

- ◆构造二叉树（* 前面2种反序列化方法，已讲）
- ◆销毁二叉树（*，已讲）
- ◆求结点数
- ◆求叶子结点数
- ◆求二叉树高度
- ◆二叉树的复制
- ◆二叉树中查找结点
- ◆判断两棵树是否相同（镜像）
- ◆删除二叉树叶子结点
- ◆判断二叉树是否严格二叉树

除*外其它操作在实践课有对应题目（请自学）

求结点数

```
template  <class DataType>
int BiTree<DataType>::recursive_size(BiNode<DataType> *sub_root) {

}


 template  <class DataType>
 int BiTree<DataType>::size() {
         return recursive_size(root);

 }
```

求结点数

```
template  <class DataType>
int BiTree<DataType>::recursive_size(BiNode<DataType> *sub_root) {
        if (sub_root == nullptr)
                return 0;
        return 1+recursive_size(sub_root->lchild) +
                        recursive_size(sub_root->rchild);
}

 template  <class DataType>
 int BiTree<DataType>::size() {
         return recursive_size(root);

 }
```

求叶子结点数

```
template  <class DataType>
int BiTree<DataType>::recursive_leafsize(BiNode<DataType> *sub_root) {



}
```

求叶子结点数

```
template  <class DataType>
int BiTree<DataType>::recursive_leafsize(BiNode<DataType> *sub_root) {
        if (sub_root == nullptr)
                return 0;
        if (sub_root->lchild == nullptr && sub_root->rchild == nullptr)
                return 1;
        return recursive_leafsize(sub_root->lchild) +
                        recursive_leafsize(sub_root->rchild);
}
```

求二叉树的高度

```
template <class DataType>
int CTree<DataType>::recursive_height1(CSNode<DataType> *sub_root) {
        if (not sub_root)
                return 0;
        int maxHeight = 0;
        CSNode<DataType> *p = sub_root->firstchild;
        while (p) {
                int h = recursive_height1(p);
                if (h > maxHeight)
                        maxHeight = h;
                p = p->rightsib;

        }
        return maxHeight + 1;
}
```

二叉树的复制

```
template <class DataType>
BiNode<DataType> *BiTree<DataType> ::recursive_copy(BiNode<DataType> * &sub_root){


}
```

```
 template <class DataType>
 BiTree<DataType> ::BiTree(BiTree<DataType> &source){


 }
```

二叉树的复制

```
template <class DataType>
BiNode<DataType> *BiTree<DataType> ::recursive_copy(BiNode<DataType> * &sub_root){

        if (sub_root == nullptr) return nullptr;
                BiNode<DataType> *temp = new BiNode<DataType>(sub_root->data);
                temp->lchild = recursive_copy(sub_root->lchild);
                temp->rchild = recursive_copy(sub_root->rchild);
                return temp;


}

template <class DataType>
BiTree<DataType> ::BiTree(BiTree<DataType> &source){
        root = recursive_copy(source.root);
}
```

二叉树下查找值为x的结点的指针（一般认为x唯一）

```
template <class DataType>
BiNode<DataType> *BiTree<DataType>::find_node(BiNode<DataType> *sub_root,
const DataType &x) const {

}

template <class DataType>
BiNode<DataType> *BiTree<DataType>::find_node(const DataType &x) const {
        return find_node(root, x);
}
```

二叉树下查找值为x的结点

```
template <class DataType>
BiNode<DataType> *BiTree<DataType>::find_node(BiNode<DataType> *sub_root, const DataType &x)
const {
        if (sub_root == nullptr || sub_root->data == x)
                return sub_root;
        else {
                BiNode<DataType> *temp = find_node(sub_root->lchild, x);
                if (temp != nullptr) return temp;
                else return find_node(sub_root->rchild, x);
                }
}

template <class DataType>
BiNode<DataType> *BiTree<DataType>::find_node(const DataType &x) const {
        return find_node(root, x);
}
```

判断两棵二叉树是否相同

- 判断分别以sub_root1和sub_root2为根的两棵二叉树是否相同，即判断它们的结构和对应值是否完全相同。

情形	1	2	3	4
sub_root1	空	非空	空	非空
sub_root2	空	空	非空	非空
是否相同	相同	不同	不同	由根结点的值、左子树和右子树是否分别相同共同决定。

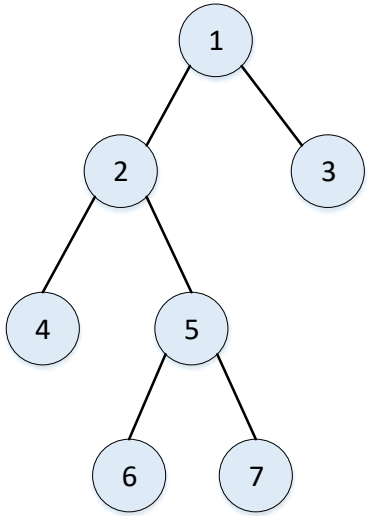
判断两棵二叉树是否相同

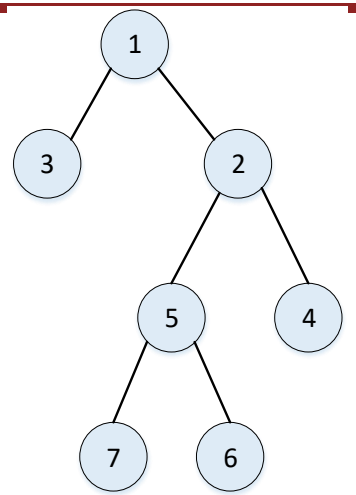
```
template <class DataType>
bool BiTree<DataType>::recursive_eq(BiNode<DataType> *sub_root1, BiNode<DataType> *sub_root2)
{
        if (sub_root1 == nullptr && sub_root2 == nullptr)
                return true; // 情形1
        if (!(sub_root1 && sub_root2))
                return false;    // 情形2、3
        // 以下为情形4
        if (sub_root1->data != sub_root2->data)
                return false;
        return recursive_eq(sub_root1->lchild, sub_root2->lchild) &&
                recursive_eq(sub_root1->rchild, sub_root2->rchild);
}
```

```
template <class DataType>
bool BiTree<DataType>::equal(BiTree<DataType> other) {
        return recursive_eq(root, other.root);
}
```

判断两棵二叉树是否互为镜像

- 判断分别以root1和root2为根的两棵二叉树是否互为镜像。



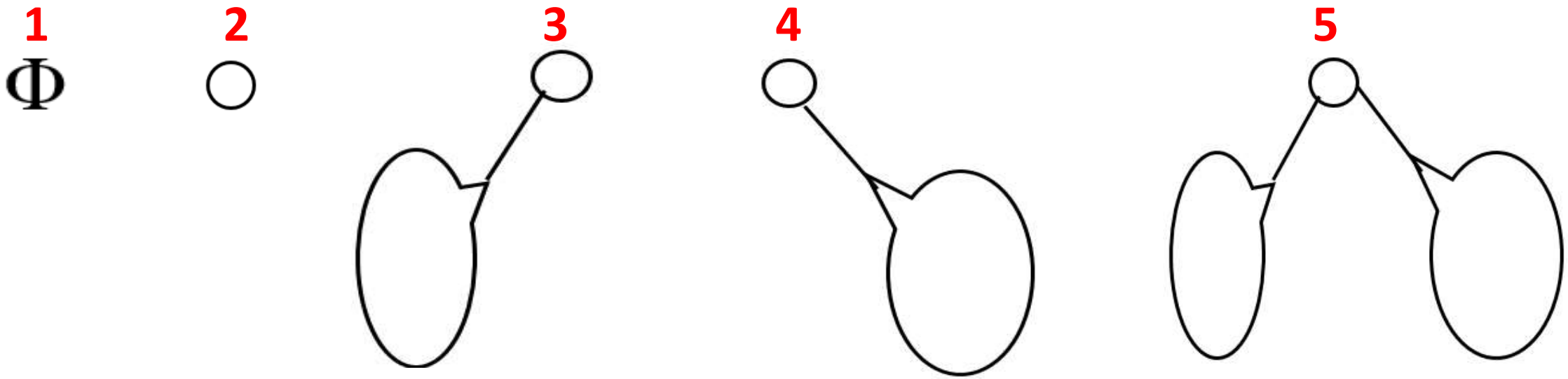


情形	1	2	3
root1	空	非空	空
root2	空	空	非空
是否互为镜像	是	否	否

4	根结点的值不同	根结点的值相同
root1和root2都非空	否	root1的左子树与root2的右子树互为镜像 且 root1的右子树与root2的左子树互为镜像

删除二叉树中当前的叶结点

- **删除以sub_root为根的二叉树中当前的叶结点，并返回新生成二叉树的根结点。**
  - 如果sub_root为空，无须删除，返回sub_root；
  - 如果sub_root无左右子树，根结点为叶子，删除该根结点后，返回None；
  - 否则，分别调用递归算法自身，删除sub_root左右子树上的叶子，并将sub_root的left和right指针分别指向删除叶子之后的新左右子树的根，最后返回根sub_root。



判断二叉树是否严格二叉树

- 递归定义：
  - 形态1，形态2，则返回true ；
  - 形态3，形态4，则返回false；
  - 形态5，由左右子树是否同时为严格二叉树决定整棵二叉树是否为严格二叉树。

1 2 3 4 5